AD-A252 471

Form Approved
OPM No.

1. AGENCY USE (Leave	2. REPORT	3. REPORT TYPE AND DATES
		Final: 18 Mar 1992 to 01 Jun 1993

4. TITLE AND

Validation Summary Report: Tartan, Inc., Tartan Ada SPARC C30 version 4.2, Sun SPARCstation/ELC (Host) to Texas Instruments TMS320C30 (Target), 920313I1.11244

5. FUNDING

6.

IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND

IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION

IABG-VSR 83

9. SPONSORING/MONITORING AGENCY NAME(S) AND

Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

DTIC
ELECTE
JUL 0 1 1992
S
A
D

10. SPONSORING/MONITORING AGENCY

11. SUPPLEMENTARY

12a. DISTRIBUTION/AVAILABILITY	12b. DISTRIBUTION
Approved for public release; distribution unlimited.	

13. *(Maximum 200*

Tartan, Inc., Tartan Ada SPARC C30 version 4.2, Sun SPARCstation/ELC (Host) to Texas Instruments TMS320C30 (Target), ACVC 1.11.

92-17195

92

14. SUBJECT

Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A,

15. NUMBER OF

16. PRICE

17. SECURITY CLASSIFICATION UNCLASSIFIED	18. SECURITY UNCLASSIFED	19. SECURITY CLASSIFICATION UNCLASSIFIED	20. LIMITATION OF

NSN

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 92031311.11244
Tartan, Inc.
Tartan Ada SPARC C30 version 4.2
Sun SPARCstation/ELC =>
Texas Instruments TMS320C30

Accesion For		
NTIS CRA&I	☑	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and/or Special	
A-1		

DTIC COPY INSPECTED

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
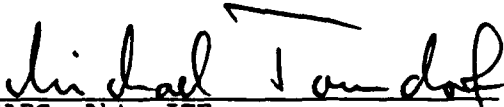Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 13 March, 1992.

Compiler Name and Version:	**Tartan Ada SPARC C30 version 4.2**
Host Computer System:	**Sun SPARCstation/ELC** **under SunOS Version 4.1.1**
Target Computer System:	**Texas Instruments TMS320C30 Application Board** **(bare machine)**

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920313I1.11244 is awarded to Tartan, Inc. This certificate expires 24 months after ANSI approval of MIL-STD 1815B.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Tartan, Inc.

Certificate Awardee: Tartan, Inc.

Ada Validation Facility: IABG mbH

ACVC Version: 1.11
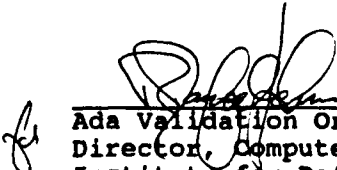
Ada Implementation:

Ada Compiler Name and Version: _Tartan Ada SPARC C30 version 4.2_

Host Computer System: _SPARC Station/ELC SunOS version 4.1.1_

Target Computer System: Texas Instruments TMS320C30 _Application Board (bare machine)_

Declaration:

I, the undersigned, declare that I have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

[signature]   _March 16, 1993_

Customer Signature   **Date**

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

>   National Technical Information Service
>   5285 Port Royal Road
>   Springfield VA   22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

>   Ada Validation Organization
>   Computer and Software Engineering Division
>   Institute for Defense Analyses
>   1801 North Beauregard Street
>   Alexandria VA   22311-1772

1.2 REFERENCES

[Ada83]   Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90]   Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89]   Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler   The software and any needed hardware that have to be added
                  to a given host and target computer system to allow
                  transformation of Ada programs into executable form and
                  execution thereof.

Ada Compiler   The means for testing compliance of Ada implementations,
Validation   consisting of the test suite, the support programs, the ACVC
Capability   user's guide and the template for the validation summary
(ACVC)   report.

Ada   An Ada compiler with its host computer system and its
Implementation   target computer system.

Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].

Validation   The process of checking the conformity of an Ada compiler to
                    the Ada programming language and of issuing a certificate
                    for this implementation.

Withdrawn   A test found to be incorrect and not used in conformity
test   testing. A test may be incorrect because it has an invalid
                    test objective, fails to meet its test objective, or
                    contains erroneous or illegal use of the Ada programming
                    language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for
withdrawing each test is available from either the AVO or the AVF. The
publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant
for a given Ada implementation. Reasons for a test's inapplicability may
be supported by documents issued by the ISO and the AJPO known as Ada
Commentaries and commonly referenced in the format AI-ddddd. For this
implementation, the following tests were determined to be inapplicable for
the reasons indicated; references to Ada Commentaries are included as
appropriate.

The following 285 tests have floating-point type declarations
requiring more digits than SYSTEM.MAX_DIGITS:

C24113F..Y (20 tests)		C35705F..Y (20 tests)	
C35706F..Y (20 tests)		C35707F..Y (20 tests)	
C35708F..Y (20 tests)		C35802F..Z (21 tests)	
C45241F..Y (20 tests)		C45321F..Y (20 tests)	
C45421F..Y (20 tests)		C45521F..Z (21 tests)	
C45524F..Z (21 tests)		C45621F..Z (21 tests)	
C45641F..Y (20 tests)		C46012F..Z (21 tests)	

The following 21 tests check for the predefined type SHORT_INTEGER; for this implementation, there is no such type:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C455C4E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

A35801E checks that FLOAT'FIRST..FLOAT'LAST may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for 'SMALL that are not powers of two or ten; this implementation does not support such values for 'SMALL.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C instantiate generic units before their bodies are compiled; this implementation creates a dependence on generic units as allowed by AI-0408 & AI-0506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (see 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation

does not support such sizes.

CD2B15B checks that STORAGE_ERROR is raised when the storage size specified for a collection is too small to hold a single value of the designated type; this implementation allocates more space than was specified by the length clause, as allowed by AI-00558.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A expect that NAME_ERROR is raised when an attempt is made to create a file with an illegal name; this implementation does not support the creation of external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see Section 1.3) were required for 106 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24007A	B24009A	B25002B	B32201A	B33204A
B33205A	B35701A	B36171A	B36201A	B37101A	B37102A
B37201A	B37202A	B37203A	B37302A	B38003A	B38003B
B38008A	B38008B	B38009A	B38009B	B38103A	B38103B
B38103C	B38103D	B38103E	B43202C	B44002A	B48002A
B48002B	B48002D	B48002E	B48002G	B48003E	B49003A
B49005A	B49006A	B49006B	B49007A	B49007B	B49009A
B4A010C	B54A20A	B54A25A	B58002A	B58002B	B59001A
B59001C	B59001I	B62006C	B67001A	B67001B	B67001C
B67001D	B74103E	B74104A	B74307B	B83E01A	B85007C
B85008G	B85008H	B91004A	B91005A	B95003A	B95007B
B95031A	B95074E	BA1001A	BC1002A	BC1109A	BC1109C
BC1206A	BC2001E	BC3005B	BD2A06A	BD2B03A	BD2D03A
BD4003A	BD4006A	BD8003A			

E28002B was graded inapplicable by Evaluation and Test Modification as directed by the AVO. This test checks that pragmas may have unresolvable arguments, and it includes a check that pragma LIST has the required effect; but, for this implementation, pragma LIST has no effect if the compilation results in errors or warnings, which is the case when the test is processed without modification. This test was also processed with the pragmas at lines 46, 58, 70 and 71 commented out so that pragma LIST had effect.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

Tests C45524A..E (5 tests) were graded passed by Test Modification as directed by the AVO. These tests expect that a repeated division will result in zero; but the Ada standard only requires that the result lie in the smallest safe interval. Thus, the tests were modified to check that the result was within the smallest safe interval by adding the following code after line 141; the modified tests were passed:

        ELSIF VAL <= F'SAFE_SMALL THEN COMMENT ("UNDERFLOW SEEMS GRADUAL");

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package report's body, and thus the packages' calls to function Report.Ident_Int at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

B83E01B was graded passed by Evaluation Modification as directed by the AVO. This test checks that a generic subprogram's formal parameter names (i.e. both generic and subprogram formal parameter names) must be distinct; the duplicated names within the generic declarations are marked as errors, whereas their recurrences in the subprogram bodies are marked as "optional" errors--except for the case at line 122, which is marked as an error. This implementation does not additionally flag the errors in the bodies and thus the expected error at line 122 is not flagged. The AVO ruled that the implementation's behavior was acceptable and that the test need not be split (such a split would simply duplicate the case in B83E01A at line 15).

CA2009A, CA2009C..D (2 tests), CA2009F and BC3009C were graded inapplicable by Evaluation Modification as directed by the AVO. These tests instantiate generic units before those units' bodies are compiled; this implementation creates dependences as allowed by AI-00408 & AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete, and the objectives of these tests cannot be met.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 & AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by compiling the seperate files in the following order (to allow re-compilation of obsolete units), and all intended errors were then detected by the compiler:

BC3204C: C0, C1, C2, C3M, C4, C5, C6, C3M

BC3205D: D0, D1M, D2, D1M

BC3204D and BC3205C were graded passed by Test Modification as directed by the AVO. These tests are similar to BC3204C and BC3205D above, except that all compilation units are contained in a single compilation. For these two tests, a copy of the main procedure (which later units make obsolete) was appended to the tests; all expected errors were then detected.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-ten value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

AD9001B and AD9004A were graded passed by Processing Modification as directed by the AVO. These tests check that various subprograms may be interfaced to external routines (and hence have no Ada bodies). This implementation requires that a file specification exists for the foreign subprogram bodies. The following command was issued to the Librarian to inform it that the foreign bodies will be supplied at link time (as the bodies are not actually needed by the program, this command alone is sufficient):

         interface -sys -L=library ad9001b & ad9004a

CE2103A, CE2103B and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical information about this Ada implementation, contact:

> Mr Ron Duursma
> Director of Ada Products
> Tartan, Inc.
> 300 Oxford Drive
> Monroeville, PA 15146
> USA
> Tel. (412) 856-3600

For sales information about this Ada implementation, contact:

> Ms. Marlyse Bennett
> Director of Sales
> Tartan, Inc.
> 12110 Sunset Hills Road
> Suite 450
> Reston, VA 22090
> USA
> Tel. (703) 715-3044

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a

floating-point precision that exceeds the implementation's maximum
precision (item e; see section 2.2), and those that depend on the support
of a file system -- if none is supported (item d). All tests passed, except
those that are listed in sections 2.1 and 2.2 (counted in items b and f,
below).

a)	Total Number of Applicable Tests	3441
b)	Total Number of Withdrawn Tests	95
c)	Processed Inapplicable Tests	85
d)	Non-Processed I/O Tests	264
e)	Non-Processed Floating-Point Precision Tests	285
f)	Total Number of Inapplicable Tests	634 (c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170 (a+b+f)

3.3 TEST EXECUTION

A magnetic data cartridge containing the customized test suite (see section 1.3)
was taken on-site by the validation team for processing. The contents of
the magnetic data cartridge were loaded directly onto the host computer.

The tests were compiled and linked on the host computer system, as
appropriate. The executable images were transferred to the target computer
system by the communications link, an RS232 Interface, and run. The results
were captured on the host computer system.

Testing was performed using command scripts provided by the customer and
reviewed by the validation team. See Appendix B for a complete listing of
the processing options for this implementation. It also indicates the
default options. The options invoked explicitly for validation testing
during this test were for compiling:

-f   forces the compiler to accept an attempt to compile a unit
        imported from another library which is normally prohibited.

-c   suppresses the creation of a registered copy of the source
        code in the library directory for use by the REMAKE and MAKE
        subcommands.

-La   forces a listing to be produced, default is to only produce a
        listing when an error occurs.

    No explicit Linker options were used.

Test output, compiler and linker listings, and job logs were captured on
magnetic data cartridge and archived at the AVF. The listings examined on-site
by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC.
The meaning and purpose of these parameters are explained in [UG89]. The
parameter values are presented in two tables. The first table lists the
values that are defined in terms of the maximum input-line length, which is
the value for $MAX_IN_LEN--also listed here. These values are expressed
here as Ada string aggregates, where "V" represents the maximum input-line
length.

Macro Parameter	Macro Value
$MAX_IN_LEN	240
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

A-1

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	1
$COUNT_LAST	2147483646
$DEFAULT_MEM_SIZE	16777216
$DEFAULT_STOR_UNIT	32
$DEFAULT_SYS_NAME	TI320C30
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.ADDRESS'(16#809803#)
$ENTRY_ADDRESS1	SYSTEM.ADDRESS'(16#809804#)
$ENTRY_ADDRESS2	SYSTEM.ADDRESS'(16#809805#)
$FIELD_LAST	240
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.50282E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E+38
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E+38
$HIGH_PRIORITY	100
$ILLEGAL_EXTERNAL_FILE_NAME1	ILLEGAL_EXTERNAL_FILE_NAME1
$ILLEGAL_EXTERNAL_FILE_NAME2	ILLEGAL_EXTERNAL_FILE_NAME2
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1

$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006F1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	TI_C
$LESS_THAN_DURATION	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST	
	-131_073.0
$LINE_TERMINATOR	' '
$LOW_PRIORITY	10
$MACHINE_CODE_STATEMENT	
	Two_Opnds'(LDI,(Imm,5),(Reg,R0));
$MACHINE_CODE_TYPE	Instruction_Mnemonic
$MANTISSA_DOC	31
$MAX_DIGITS	9
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648
$NAME	NO_SUCH_TYPE_AVALIABLE
$NAME_LIST	TI320C30
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	16777216
$NEW_STOR_UNIT	32
$NEW_SYS_NAME	TI320C30
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	record Operation: Instruction_Mnemonic; Operand_1: Operand; end record;
$RECORD_NAME	Two_Opnds
$TASK_SIZE	32
$TASK_STORAGE_SIZE	4096
$TICK	0.00006103515625
$VARIABLE_ADDRESS	SYSTEM.ADDRESS'(16#809800#)
$VARIABLE_ADDRESS1	SYSTEM.ADDRESS'(16#809801#)
$VARIABLE_ADDRESS2	SYSTEM.ADDRESS'(16#809802#)

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described
in this Appendix, are provided by the customer. Unless specifically noted
otherwise, references in this appendix are to compiler documentation and
not to this report.

Chapter 4
Compiling Ada Programs

The tadac30 command is used to compile and assemble Ada compilation units.

4.1. THE tadac30 COMMAND FORMAT

The tadac30 command has this format:

    % tadac30 [option...] file... [option...]

Arguments that start with a hyphen are interpreted as options; otherwise, they represent filenames. There must be at least one filename, but there need not be any options. Options and filenames may appear in any order, and all options apply to all filenames. For an explanation of the available options, see Section 4.2.

If a source file does not reside in the directory in which the compilation takes place, the *file* must include a path sufficient to locate the file. It is recommended that only one compilation unit be placed in a file.

If no extension is supplied with the file name, a default extension of .ada will be supplied by the compiler.

Files are processed in the order in which they appear on the command line. The compiler sequentially processes all compilation units in each file. Upon successful compilation of a unit:

- The library database, Lbrry.Db, is updated with the new compilation time and any new dependencies.
- One or more separate compilation files and/or object files are generated.

If no errors are detected in a compilation unit, tadac30 produces an object module and updates the library. If any error is detected, no object code file is produced, a source listing is produced, and no library entry is made for that compilation unit. If warnings are generated, both an object code file and a source listing are produced. For further details about the process of updating the library, files generated, replacement of existing files, and possible error conditions, see Sections 4.3 through 4.5.

The output from tadac30 is a file of type .stof or .tof, for a specification or a body unit respectively, containing object code. Some other files are generated as well. See Section 4.4 for a list of extensions of files that may be generated.

The compiler is capable of limiting the number of library units that become obsolete by recognizing *refinements*. A library unit is a refinement of its previously compiled version if the only changes that were made are:

- Addition or deletion of comments.
- Addition of subprogram specifications after the last declarative item in the previous version.

An option is required to cause the compiler to detect refinements. When a refinement is detected by the compiler, dependent units are not marked as obsolete.

4.2. OPTIONS

Command line options indicate special actions to be performed by the compiler or special output file properties.

The following command line options may be used:

	the runtimes. Please contact Tartan for information on how to perform these customizations.
-ndb	Controls whether the compiler generates delayed branch instructions (detailed in Section 5.12).
-nhl	When the nhl (no huge loops) qualifier is specified, the user is asserting that no loops will iterate more than 2^{23} times. This limit includes non-user specific loops, such as those generated by the compiler to operate on large objects. Erroneous code will be generated if this assertion is false. The compiler will generate a run-time range check if overflow checks are not suppressed.
-Opn	Control the level of optimization performed by the compiler, requested by n. The optimization levels available are:

$n = 0$ **Minimum** - Performs context determination, constant folding, algebraic manipulation, and short circuit analysis.

$n = 1$ **Low** - Performs level 0 optimizations plus common subexpression elimination and equivalence propagation within basic blocks. It also optimizes evaluation order.

$n = 2$ **Best tradeoff for space/time** - *the default level*. Performs level 1 optimizations plus flow analysis which is used for common subexpression elimination and equivalence propagation across basic blocks. It also performs invariant expression hoisting, dead code elimination, and assignment killing. Level 2 also performs lifetime analysis which is used to improve register allocation. It also performs inline expansion of subprogram calls indicated by pragma INLINE, if possible.

$n = 3$ **Time** - Performs level 2 optimizations plus inline expansion of subprogram calls which the optimizer decides are profitable to expand (from an execution time perspective). Other optimizations which improve execution time at a cost to image size are performed only at this level.

$n = 4$ **Space** - Performs those optimizations which usually produce the smallest code, often at the expense of speed. Please note that this optimization level may not always produce the smallest code. Under certain conditions another level may produce smaller code.

-p	Extracts syntactically correct compilation unit source from the parsed file and loads this file into the library as a parsed unit. Parsed units are, by definition, inconsistent. This switch allows users to load units into the library without regard to correct compilation order. The command remakecu is used subsequently to reorder the compilation units in the correct sequence. See Section 10.2.5 for a more complete description of this command.
-r	*Data on this switch is provided for* information only. This switch is used exclusively by the librarian to notify the compiler that the source undergoing compilation is an internal source file. The switch causes the compiler to retain old external source file information. This switch should be used only by the librarian and command files created by the librarian. See Section 3.6.1.
-S[ACDEILORSZ]	Suppress the given set of checks:

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-
dependent pragmas, to certain machine-dependent conventions as mentioned
in Chapter 13 of the Ada Standard, and to certain allowed restrictions on
representation clauses. The implementation-dependent characteristics of
this Ada implementation, as described in this Appendix, are provided by
the customer. Unless specifically noted otherwise, references in this
Appendix are to compiler documentation and not to this report.
Implementation-specific portions of the package STANDARD, are outlined
below for convenience.

```
package STANDARD is

    ...

   type INTEGER is range -2147483648 .. 2147483647;

   type FLOAT is digits 6 range
      -2#1.00000000000000000000000#e+128 ..
       2#0.11111111111111111111111#e+128;

   type LONG_FLOAT is digits 9 range
      -2#1.0000000000000000000000000000000#e+128 ..
       2#0.1111111111111111111111111111111#e+128;

   type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

    ...

end STANDARD;
```

Chapter 5
Appendix F to MIL-STD-1815A

This chapter contains the required Appendix F to the LRM which is *Military Standard, Ada Programming Language*, ANSI/MIL-STD-1815A (American National Standards Institute, Inc., February 17, 1983).

5.1. PRAGMAS

5.1.1. Predefined Pragmas

This section summarizes the effects of and restrictions on predefined pragmas.

- Access collections are not subject to automatic storage reclamation so pragma CONTROLLED has no effect. Space deallocated by means of UNCHECKED_DEALLOCATION will be reused by the allocation of new objects.

- Pragma ELABORATE is supported.

- Pragma INLINE is supported.

- Pragma INTERFACE is supported. The Language_Name TI_C is used to make calls to subprograms (written in the Texas Instruments *C* language) from Tartan Ada. Any other Language_Name will be accepted, but ignored, and the default will be used.

- Pragma LIST is supported but has the intended effect only if the command line option -La was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.

- Pragma MEMORY_SIZE is supported. See Section 5.1.3.

- Pragma OPTIMIZE is supported except when at the outer level (that is, in a package specification or body).

- Pragma PACK is supported.

- Pragma PAGE is supported but has the intended effect only if the command line option -La was supplied for compilation, and the listing generated was not due to the presence of errors and/or warnings.

- Pragma PRIORITY is supported.

- Pragma STORAGE_UNIT is accepted but no value other than that specified in package System (Section 5.3) is allowed.

- Pragma SHARED is not supported.

- Pragma SUPPRESS is supported.

- Pragma SYSTEM_NAME is accepted but no value other than that specified in package System (Section 5.3) is allowed.

5.1.2. Implementation-Defined Pragmas

Implementation-defined pragmas provided by Tartan are described in the following sections.

The foreign body is entirely responsible for initializing objects declared in a package utilizing pragma FOREIGN_BODY. In particular, the user should be aware that the implicit initializations described in LRM 3.2.1 are not done by the compiler. (These implicit initializations are associated with objects of access types, certain record types and composite types containing components of the preceding kinds of types.)

Pragma LINKAGE_NAME should be used for all declarations in the package, including any declarations in a nested package specification to be sure that there are no conflicting link names. If pragma LINKAGE_NAME is not used, the cross-reference qualifier, -x, (see Section 4.2) should be used when invoking the compiler and the resulting cross-reference table of linknames inspected to identify the linknames assigned by the compiler and determine that there are no conflicting linknames (see also Section 4.5.4.2). In the following example, we want to call a function plmn which computes polynomials and is written in C.

```
package Math_Functions is
   pragma FOREIGN_BODY ("C");
   function POLYNOMIAL   (X:INTEGER) return INTEGER;
           -- Ada spec matching the C routine
   pragma LINKAGE_NAME 'POLYNOMIAL, "plmn");
           -- Force compiler to use name "plmn" when referring to this
           -- function
           -- Note: The linkage name "plmn" may need to be "_plmn",
           --        if the C compiler produces leading underscores
           --        for external symbols.
end Math_Functions;

with Math_Functions; use Math_Functions;
procedure MAIN is
   X:INTEGER := POLYNOMIAL(10);
           -- Will generate a call to "plmn"
   begin ...
end MAIN;
```

To compile, link and run the above program, you do the following steps:

1. Compile Math_Functions

2. Compile MAIN

3. Provide the object module (for example, math.tof) containing the compiled "C" code for plmn.

4. Issue the command:

```
% adalibc30 foreign Math_Functions math.tof
```

5. Issue the command:

```
% adalibc30 link main
```

Without Step 4, an attempt to link will produce an error message informing you of a missing package body for Math_Functions.

Using an Ada body from another Ada program library. The user may compile a body written in Ada for a specification into the library, regardless of the language specified in the pragma contained in the specification. This capability is useful for rapid prototyping, where an Ada package may serve to provide a simulated response for the functionality that a foreign body may eventually produce. It also allows the user to replace a foreign body with an Ada body without recompiling the specification.

The user can either compile an Ada body into the library, or use the command adalibc30 foreign (see Section 3.3.3) to use an Ada body from another library. The Ada body from another library must have been compiled under an identical specification. The pragma LINKAGE_NAME must have been applied to all entities declared in the specification. The only way to specify the linkname for the elaboration routine of an Ada body is with the pragma FOREIGN_BODY.

5.4.2. Length Clauses

Length clauses (LRM 13.2) are, in general, supported. The following sections detail use and restrictions.

5.4.2.1. Size Specifications for Types

The rules and restrictions for size specifications applied to types of various classes are described below.

The following principle rules apply:

1. The size is specified in bits and must be given by a static expression.

2. The specified size is taken as a mandate to store objects of the type in the given size wherever feasible. No attempt is made to store values of the type in a smaller size, even if possible. The following rules apply with regard to feasibility:

   - An object that is not a component of a composite object is allocated with a size and alignment that is referable on the target machine; that is, no attempt is made to create objects of non-referable size on the stack. If such stack compression is desired, it can be achieved by the user by combining multiple stack variables in a composite object; for example:

     ```
     type My_Enum is (A,B);
     for My_enum'size use 1;
     V,W: My_enum; -- will occupy two storage
                   -- units on the stack
                   -- (if allocated at all)
     type rec is record
        V,W: My_enum;
     end record;
     pragma PACK(rec);
     O: rec;        -- will occupy one storage unit
     ```

   - A formal parameter of the type is sized according to calling conventions rather than size specifications of the type. Appropriate size conversions upon parameter passing take place automatically and are transparent to the user.

   - Adjacent bits to an object that is a component of a composite object, but whose size is non-referable, may be affected by assignments to the object, unless these bits are occupied by other components of the composite object; that is, whenever possible, a component of non-referable size is made referable.

   In all cases, the compiler generates correct code for all operations on objects of the type, even if they are stored with differing representational sizes in different contexts.

   Note: A size specification cannot be used to force a certain size in value operations of the type; for example:

   ```
   type my_int is range 0..65535;
   for my_int'size use 16; -- o.k.
   A,B: my_int;
       ...A + B... -- this operation will generally be
                   -- executed on 32-bit values
   ```

3. A size specification for a type specifies the size for objects of this type and of all its subtypes. For components of composite types, whose subtype would allow a shorter representation of the component, no attempt is made to take advantage of such shorter representations. In contrast, for types without a length clause, such components may be represented in a lesser number of bits than the number of bits required to represent all values of the type. For example:

A size specification cannot be applied to a record type with components of dynamically determined size.

Note: Size specifications for records can be used only to widen the representation accomplished by padding at the beginning or end of the record. Any narrowing of the representation over default type mapping must be accomplished by representation clauses or pragma PACK.

5.4.2.5. Specification of Collection Sizes

The specification of a collection size causes the collection to be allocated with the specified size. It is expressed in storage units and need not be static; refer to package System for the meaning of storage units.

Any attempt to allocate more objects than the collection can hold causes a STORAGE_ERROR exception to be raised. Dynamically sized records or arrays may carry hidden administrative storage requirements that must be accounted for as part of the collection size. Moreover, alignment constraints on the type of the allocated objects may make it impossible to use all memory locations of the allocated collection. No matter what the requested object size, the allocator must allocate a minimum of 2 words per object. This lower limit is necessary for administrative overhead in the allocator. For example, a request of 5 words results in an allocation of 5 words; a request of 1 word results in an allocation of 2 words.

In the absence of a specification of a collection size, the collection is extended automatically if more objects are allocated than possible in the collection originally allocated with the compiler-established default size. In this case, STORAGE_ERROR is raised only when the available target memory is exhausted. If a collection size of zero is specified, no access collection is allocated.

5.4.2.6. Specification of Task Activation Size

The specification of a task activation size causes the task activation to be allocated with the specified size. It is expressed in storage units; refer to package System for the meaning of storage units.

Any attempt to exceed the activation size during execution causes a STORAGE_ERROR exception to be raised. Unlike collections, there is no extension of task activations.

5.4.2.7. Specification of ' SMALL

Only powers of 2 are allowed for ' SMALL.

The length of the representation may be affected by this specification. If a size specification is also given for the type, the size specification takes precedence; it must then be possible to accommodate the specification of ' SMALL within the specified size.

5.4.3. Enumeration Representation Clauses

For enumeration representation clauses (LRM 13.3), the following restrictions apply:

- The internal codes specified for the literals of the enumeration type may be any integer value between INTEGER' FIRST and INTEGER' LAST. It is strongly advised to not provide a representation clause that merely duplicates the default mapping of enumeration types, which assigns consecutive numbers in ascending order starting with 0, since unnecessary runtime cost is incurred by such duplication. It should be noted that the use of attributes on enumeration types with user-specified encodings is costly at run time.

- Array types, whose index type is an enumeration type with non-contiguous value encodings, consist of a contiguous sequence of components. Indexing into the array involves a runtime translation of the index value into the corresponding position value of the enumeration type.

5.4.4. Record Representation Clauses

The alignment clause of record representation clauses (LRM 13.4) is observed.

Static objects may be aligned at powers of 2. The specified alignment becomes the minimum alignment of the record type, unless the minimum alignment of the record forced by the component allocation and the minimum alignment requirements of the components is already more stringent than the specified alignment.

It should be noted that the default type mapping for records maps components of boolean or other types that require only a single bit to a single bit in the record layout, if there are multiple such components in a record. Otherwise, it allocates a referable amount of storage to the component.

If pragma PACK is applied to a record for which a record representation clause has been given detailing the allocation of some but not all components, the pragma PACK affects only the components whose allocation has not been detailed. Moreover, the strategy of not utilizing gaps between explicitly allocated components still applies.

5.4.7. Minimal Alignment for Types

Certain alignment properties of values of certain types are enforced by the type mapping rules. Any representation specification that cannot be satisfied within these constraints is not obeyed by the compiler and is appropriately diagnosed.

Alignment constraints are caused by properties of the target architecture, most notably by the capability to extract non-aligned component values from composite values in a reasonably efficient manner. Typically, restrictions exist that make extraction of values that cross certain address boundaries very expensive, especially in contexts involving array indexing. Permitting data layouts that require such complicated extractions may impact code quality on a broader scale than merely in the local context of such extractions.

Instead of describing the precise algorithm of establishing the minimal alignment of types, we provide the general rule that is being enforced by the alignment rules:

- No object of scalar type including components or subcomponents of a composite type, may span a target-dependent address boundary that would mandate an extraction of the object's value to be performed by two or more extractions.

5.5. IMPLEMENTATION-GENERATED COMPONENTS IN RECORDS

The only implementation-dependent components allocated by Tartan Ada in records contain dope information for arrays whose bounds depend on discriminants of the record. These components cannot be named by the user.

5.6. INTERPRETATION OF EXPRESSIONS APPEARING IN ADDRESS CLAUSES

Section 13.5.1 of the Ada Language Reference Manual describes a syntax for associating interrupts with task entries. Tartan Ada implements the address clause

```
for toentry use at intID;
```

by associating the interrupt specified by intID with the toentry entry of the task containing this address clause. The interpretation of intID is both machine and compiler dependent.

5.7. RESTRICTIONS ON UNCHECKED CONVERSIONS

Tartan supports UNCHECKED_CONVERSION as documented in Section 13.10 of the Ada Language Reference Manual. The sizes need not be the same, nor need they be known at compile time. If the value in the source is wider than that in the target, the source value will be truncated. If narrower, it will be zero-extended. Calls on instantiations of UNCHECKED_CONVERSION are made inline automatically.

5.8. IMPLEMENTATION-DEPENDENT ASPECTS OF INPUT-OUTPUT PACKAGES

Tartan Ada supplies the predefined input/output packages DIRECT_IO, SEQUENTIAL_IO, TEXT_IO, and LOW_LEVEL_IO as required by LRM Chapter 14. However, since the 320C30 chip is used in embedded applications lacking both standard I/O devices and file systems, the functionality of DIRECT_IO, SEQUENTIAL_IO, and TEXT_IO is limited.

DIRECT_IO and SEQUENTIAL_IO raise USE_ERROR if a file open or file access is attempted. TEXT_IO is supported to CURRENT_OUTPUT and from CURRENT_INPUT. A routine that takes explicit file names raises USE_ERROR.

5.9.5. *Values of Integer Attributes*

Tartan Ada supports the predefined integer type INTEGER. The range bounds of the predefined type INTEGER are:

Attribute	Value
INTEGER'FIRST	-2**31
INTEGER'LAST	2**31-1

The range bounds for subtypes declared in package TEXT_IO are:

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST - 1
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	INTEGER'LAST - 1
FIELD'FIRST	0
FIELD'LAST	240

The range bounds for subtypes declared in packages DIRECT_IO are:

Attribute	Value
COUNT'FIRST	0
COUNT'LAST	INTEGER'LAST
POSITIVE_COUNT'FIRST	1
POSITIVE_COUNT'LAST	COUNT'LAST

Attribute	Value for LONG_FLOAT
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0#E-7 (approximately 9.31322575E-10)
SMALL	16#0.8000_0000_0#E-31 (approximately 2.35098870E-38)
LARGE	16#0.FFFF_FFFE_0#E+31 (approximately 2.12676479E+37)
SAFE_EMAX	126
SAFE_SMALL	16#0.2000_0000_0#E-31 (approximately 5.87747175E-39)
SAFE_LARGE	16#0.3FFF_FFFF_8#E+32 (approximately 8.50705917E+37)
FIRST	-16#0.1000_0000_0#E+33 (approximately -3.40282367E+38)
LAST	16#0.FFFF_FFFF_0#E+32 (approximately 3.40282367E+38)
MACHINE_RADIX	2
MACHINE_MANTISSA	32
MACHINE_EMAX	128
MACHINE_EMIN	-126
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE

5.10. *SUPPORT FOR PACKAGE* MACHINE_CODE

Package MACHINE_CODE provides the programmer with an interface through which to request the generation of any instruction that is available on the C30. The implementation of package MACHINE_CODE is similar to that described in Section 13.8 of the Ada LRM, with several added features. Please refer to Appendix B for the Package MACHINE_CODE specification.

5.10.1. Basic Information

As required by LRM, Section 13.8, a routine which contains machine code inserts may not have any other kind of statement, and may not contain an exception handler. The only allowed declarative item is a use clause. Comments and pragmas are allowed as usual.

5.10.2. Instructions

A machine code insert has the form TYPE_MARK' RECORD_AGGREGATE, where the type must be one of the records defined in package MACHINE_CODE. Package MACHINE_CODE defines seven types of records. Each has an opcode and zero to 6 operands. These records are adequate for the expression of all instructions provided by the C30.

5.10.3. Operands and Address Modes

An operand consists of a record aggregate which holds all the information to specify it to the compiler. All operands have an address mode and one or more other pieces of information. The operands correspond exactly to the operands of the instruction being generated.

Each operand in a machine code insert must have an *Address_Mode_Name*. The address modes provided in package MACHINE_CODE provide access to all address modes supported by the C30.

The next example illustrates the correction required when the displacement is out of range for the first operand of an ADDI3 instruction. The displacement is first loaded into one of the index registers.

```
Three_Opnds'(ADDI3, (IPDA, AR3, 2), (Reg, R0), (Reg, R1))
```

will produce a code sequence like

```
LDI     2, IR0
ADDI3   AR3(IR0), R0, R1
```

In -Fixup=Warn mode, the compiler will also do its best to correct any incorrect operands for an instruction. However, a warning message is issued stating that the machine code insert required additional machine instructions to make its operands legal.

5.10.6. Assumptions Made in Correcting Operands

When compiling in -Fixup=Quiet or -Fixup=Warn modes, the compiler attempts to emit additional code to move "the right bits" from an incorrect operand to a place which is a legal operand for the requested instruction. The compiler makes certain basic assumptions when performing these corrections. This section explains the assumptions the compiler makes and their implications for the generated code. Note that if you want a correction which is different from that performed by the compiler, you must make explicit machine code insertions to perform it.

For source operands:

- Symbolic_Address means that the *address* specified by the 'ADDRESS expression is used as the source bits. When the Ada object specified by the 'ADDRESS instruction is bound to a register, this will cause a compile-time error message because it is not possible to "take the address" of a register.

- Symbolic_Value means that the *value* found at the address specified by the 'ADDRESS expression will be used as the source bits. An Ada object which is bound to a register is correct here, because the contents of a register can be expressed on the C30.

- PcRel indicates that the *address* of the label will be used as the source bits.

- Any other non-register means that the *value* found at the address specified by the operand will be used as the source bits.

For destination operands:

- Symbolic_Address means that the desired destination for the operation is the *address* specified by the 'ADDRESS expression. An Ada object which is bound to a register is correct here; a register is a legal destination on the C30.

- Symbolic_Value means that the desired destination for the operations is found by fetching 32 bits from the address specified by the 'ADDRESS expression, and storing the result to the address represented by the fetched bits. This is equivalent to applying one extra indirection to the address used in the Symbolic_Address case.

- All other operands are interpreted as directly specifying the destination for the operation.

5.10.7. Register Usage

Since the compiler may need to allocate registers as temporary storage in machine code routines, there are some restrictions placed on your register usage. The compiler will automatically free all registers which are volatile across a call for your use (that is R0..R3, bits 32-39 of R4..R5, bits 0-7 of R6..R7, AR0..AR2, IR0, IR1, BK, ST, IE, IF, IOF, RS, RC, RE).

If you reference any other register, the compiler will reserve it for your use until the end of the machine code routine. The compiler will *not* save the register automatically if this routine is inline expanded. This means that the first reference to a register which is not volatile across calls should be an instruction which saves its value in a

Name	Meaning
MOVI	Move a 32-bit integer from the first operand to the second, emitting some combination of LDI and STI's to do so.
MOVF32	Move a 32-bits float from the first operand to the second, emitting some combination of LDF and STF's to do so.
MOVF40	Move a 40-bit float from the first operand to the second, emitting some combination of LDF/LDI and STF/STI's do so.

5.10.11. Unsafe Assumptions

There are a variety of assumptions which should *not* be made when writing machine code inserts. Violation of these assumptions may result in the generation of code which does not assemble or which may not function correctly.

- The compiler *will not* generate call site code for you if you emit a call instruction. You must save and restore any volatile registers which currently have values in them, etc. If the routine you call has out parameters, a large function return result, or an unconstrained result, it is your responsibility to emit the necessary instructions to deal with these constructs as the compiler expects. In other words, when you emit a call, you must follow the linkage conventions of the routine you are calling. For further details on call site code, see Sections 6.4, 6.5 and 6.6.

- Do not assume that the 'ADDRESS on Symbolic_Address or Symbolic_Value operands means that you are getting an ADDRESS to operate on. The Address- or Value-ness of an operand is determined by your choice of Symbolic_Address or Symbolic_Value. This means that to add the *contents* of X to AR0, you should write

```
Two_Opnds' (ADDI,  (Symbolic_Value, X'ADDRESS),
                   (Reg,  AR0))
```

but to add the *address* of X to AR0, you should write

```
Two_Opnds' (ADDI,  (Symbolic_Address, X'ADDRESS),
                   (Reg,  AR0));
```

5.10.12. Limitations

The current implementation of the compiler is unable to fully support automatic correction of certain kinds of operands. In particular, the compiler assumes that the size of a data object is the same as the number of bits which is operated on by the instruction chosen in the machine code insert. This means that the insert:

```
Two_Opnds' (ADDF,  (Symbolic_Value,  Long_Float_Variable'ADDRESS),
                   (Reg,  R0))
```

will not generate correct code when Long_Float_Variable is bound to memory. The compiler will assume that Long_Float_Variable is 32 bits, when in fact it is stored in 64 bits of memory. If, on the other hand, Long_Float_Variable was bound to an extended-precision register, the insertion will function properly, as no correction is needed.

Note that the use of X'ADDRESS in a machine code insert *does not* guarantee that X will be bound to memory. This is a result of the use of 'ADDRESS to provide a "typeless" method for naming Ada objects in machine code inserts. For example, it is legal to say (Symbolc_Value, X'ADDRESS) in an insert even when X is found in a register.

```
        LDIU    @DEF2,AR0               ; line 7
        LDIU    AR3,AR1
        ADDI    2,AR1
        LDIU    *AR0++(1),R1
        RPTS    2
        LDI     *AR0++(1),R1
     || STI     R1,*AR1++(1)
        STI     R1,*AR1
        LDI     *+AR3(2),R0             ; line 43
        BLT     L22
        LDIU    *+AR3(4),AR7            ; line 44
        LDFU    R6,R7
        LDIU    R6,R7
        LDIU    1,R1            ; line 18
        LDIU    IR0,AR0
        SUBI3   R1,AR7,R2
        STI     R2,*AR0
        LDI     @DEF3,R0                ; line 19
        STI     R0,*AR0
        LDI     *+AR0(IR0),R2            ; line 20
        STI     R2,*AR1
        BU      AR1             ; line 21
L14:
        .word   L15

        .word   L16

        .word   L17

L15:    LDI     0,R7            ; line 27
        BU      L18             ; line 28
L16:    ADDI    1,R7            ; line 30
        BU      L18             ; line 31
L17:    MPYI    R7,R7           ; line 33
L18:    NOP                     ; line 35
        LDIU    R7,R6           ; line 44
L22:

        LDIU    *+AR3(6),R6
        LDIU    *+AR3(7),R7
        LDIU    *+AR3(8),AR7
        LDIU    AR3,SP
        POP     AR3
        RETSU

; Total words of code in the above routine = 46

        .data
DEF3:   .word   L14
DEF1:   .word   L22


        .text

casestatement$00:       RETSU

; Total words of code in the above routine = 1

        .data


        .text
        .data
DEF2:   .word   DEF4
DEF4:   .word   1
        .word   2
```

5.12. *DELAYED BRANCHES*

A feature of the C30 architecture is the inclusion of delayed branching. Because of the processor pipelining, normal branch instructions require four cycles to execute. During that time the pipeline is emptied and no other useful instructions may be executed. However, a second set of branch instructions is provided that allow three more instructions to be executed after initiation of the branch and before actual transfer of control. It is very important to use delayed branches whenever possible in order to achieve maximum processor throughput.

5.12.1. Generating Delayed Branches

A special machine-dependent optimization phase attempts to generate delayed branches by seeking to identify instructions that can be scheduled within the three-instruction branch delay. An instruction may be scheduled during the branch delay if it is:

1. An instruction that currently precedes the branch in the basic block and produces no result or side effect that could be used by any instruction preceding its potential location within the branch delay.

2. An instruction that currently follows a conditional branch, providing it has no side effects if the branch *is* taken and produces no result or side effect that could be mis-used by any instruction between its potential branch delay location and its current location.

3. A replication of an instruction that is currently at the address of the branch destination. If the branch is conditional, this instruction must have no side effects if the branch is *not* taken. The branch target address must be changed to point to the next address if such an instruction is discovered.

Instructions which are themselves branches may *not* be scheduled within the branch delay.

As an example, in the following code fragment, as presented to the delay branch optimization, can the BEQ be usefully transformed into the delayed version, BEQD?

```
No.       Instruction

10        LDI    0,R0            ; R0 := 0;
11        ADDI   1,R1            ; R1 := R1 + 1;
12        CMPI   R1,R2           ; compare R1 to R2
13        BEQ    L1              ; branch if equal to L1
14        LDI    *-AR1(1),R3     ; R3 := some memory value
       || LDI    *AR7++(IR0),R4  ; and in parallel also load R4
15        ADDI   R7,R5           ; R5 := R5 + R7;


        . . .


21 L1:    LDI    33,R3           ; R3 := 33;
22        LDI    R4,R5           ; R4 := R5;
```

Searching for instructions in the first class above, the delayed branch optimizer discovers that instruction 10 can be moved down to the branch delay because its results are not used by instructions 11 or 12. Instruction 12 is not movable since its result is used by the conditional branch. Likewise, instruction 11's result is used by instruction 12 and so it cannot be moved.

Applying the rules for the second type of delayed branch candidate, the delayed branch optimizer discovers that instruction 14 (a two operation parallel instruction) cannot be moved up into the branch delay since it loads R4 and the contents of R4 are read by instruction 22 if the branch is taken. Instruction 15 can be moved up into the branch delay because it produces no result that can affect instruction 14 and its results are voided by instruction 22 if the branch is taken.

While searching for the members of the third class, it is detected that instruction 21 can be replicated in the branch delay and the branch retargeted, because its result is voided by instruction 14 if the branch is not taken. Instruction 22 cannot be moved since it will leave R5 in the incorrect state for instruction 15.

After the transformations are made, the code is:

address will also always be executed, so they also fall into this category. However, instructions that follow a conditional branch will only be executed if the branch is not taken, and instructions at the branch target address of a conditional branch will only be executed if the branch is taken. Therefore, it is not always beneficial to fill the delay slots with these kind of instructions. These instructions will fill a delay slot only when the resulting code is at least as fast as the orignal code regardless of whether the branch is taken or not. The exception to this is that instructions below a conditional branch will be considered as always being executed when the delay branch optimizer can determine that the condition on the branch will not be satisfied a large percentage of the time.

5.13. *PACKAGE* INTRINSICS

The Intrinsics package is provided as a means for the programmer to access certain hardware capabilities of the 320C30 in an efficient manner.

The package declares generic functions which may be instantiated to create functions that have particularly efficient implementations. A call to such a function usually does not include a hardware subroutine call at all, but is implemented inline as a few 320C30 instructions. (Often a single instruction!)

5.13.1. *Native Instructions*

The following group of generic functions allows specific 320C30 instructions to be applied to Ada entities. The user must instantiate the generic function for the types that will be used as the operand(s) and result of the operation. These generic functions have been given the same name as the assembler's name for the corresponding instruction. In some cases this convention leads to a conflict with an Ada reserved word. This conflict is resolved by using the instruction name with an "i" appended to it.

For details of the operation applied by calling an instance of one of these generic functions see the *TMS320C30 User's Guide*. Examples of their use are given in Figures 5-1 and 5-2. Refer to appendix C for the signatures of all intrinsics. The available operations are shown in the following table.

Name	Meaning
ANDi	Bitwise logical-AND
ANDN	Bitwise logical AND-NOT
ASH	Arithmetic shift
FIX	Floating point to integer conversion
FLOATi	Integer to floating point conversion
LDE	Load floating point exponent
LDM	Load floating point mantissa
LSH	Logical shift
MPYF	32-bit x 32-bit -> 40-bit floating multiply
MPYI	24-bit x 24-bit -> 32-bit integer multiply
NORM	Floating point normalize
RND	Round floating point
NOTi	Bitwise logical complement
ORi	Bitwise logical OR
ROL	Rotate left
ROR	Rotate right
SUBC	Subtract integer conditionally
XORi	Bitwise exclusive OR

Function	Meaning
No_Overflow_ADDI	Binary add with no overflow detection. Result is always same as true mathematical answer truncated to 32-bits.
No_Overflow_SUBI	Binary subtract with no overflow detection. Result is always same as true mathematical answer truncated to 32-bits.
No_Overflow_MPYI	Binary multiply with no overflow detection. Result is always same as true mathematical answer truncated to 32-bits.

5.13.3. 40-bit Floating Multiply Variants

The 40-bit floating multiply routine, the default used by the compiler, produces accurate answers across the entire floating range. However, there is a minor speed penalty for supporting accurate multiplication for operands with extremely negative machine exponents (i.e. values of -104 or less in the upper 8-bits of the float). Therefore, a fast version that drops some bits for operands with extremely negative exponents is provided through the intrinsics package. Use it whenever the floating values are not expected to contain large negative exponents.

Function	Meaning
Fast_MPYF40	40-bit floating multiply that produces accurate answers for all input values with machine exponents greater than -104. For input values with exponent values less than or equal to -104, the bottom 8 bits of the 32-bit result mantissa are suspect. When checks are suppressed, executes in 13 cycles and may be inlined automatically by the compiler to a sequence that executes in 3-11 cycles. When checks are not suppressed, executes in 15-35 cycles, with 15 being typical (35 is the "almost overflow" case).
Slow_MPYF40	40-bit floating multiply that produces accurate answers for all input values. When checks are suppressed, executes in 14-47 cycles, with 14 being typical (47 is the "almost under/overflow" case). When checks are suppressed, the multiplication may be inlined automatically by the compiler to a sequence that executes in 3-13 cycles. When checks are not suppressed, executes in 17-44 cycles, with 17 being typical (44 is the "almost under/overflow" case).

5.13.4. 40-bit Floating Divide Variants

The 40-bit floating divide routine, the default used by the compiler, produces accurate answers across the entire floating range. However, there is a minor speed penalty for supporting accurate division for operands with extremely negative machine exponents (i.e. values of -119 or less in the upper 8-bits of the float). Therefore, a fast version that drops some bits for operands with extremely negative exponents is provided through the intrinsics package. Use it whenever the floating values are not expected to contain large negative exponents.

The 320C30 hardware places certain requirements on the addresses used in circular addressing operations. The values passed to an instantiation of Init_Circ_Iter, the iterator initialization function, must obey these rules. This normally means that it is necessary to use an address clause to position the entity whose address is passed as the first parameter of Init_Circ_Iter.

The functions that operate on circular iterators are:

Name	Meaning
Init_Circ_Iter	Procedure. Allocate and initialize an iterator. From left to right, the parameters are:

EB_Plus_Start_Index
    Usually Array (Start_Index)' address. Given n such that n is smallest value where $2^{**}n$ > BK, then Array' address mod $2^{**}n$) *must* = 0, which can be guaranteed only if Array is placed in memory using an Ada address clause.

Step   Usually Array (0)' size/32.

BK   Usually Array' size/32

Name   One of Circ_Iterator_Name_Type

Name	Meaning
Release_Circ_Iter	Procedure. Releases the iterator resources to the compiler for other use.
Read_Circ_Iter	Returns the value pointed to by the current value of the iterator, plus some arbitrary integer offset. The offset is added non-circularly.
Read_Then_Circ_Add	Returns the value pointed to by the current value of the iterator, then advances the iterator in accordance with the Step and BK specified in the initialization.
Read_Then_Circ_Sub	Returns the value pointed to by the current value of the iterator, then advances the iterator in accordance with the Step and BK specified in the initialization.
Write_Circ_Iter	Procedure. Writes the location pointed to by the current value of the iterator, plus some arbitrary integer offset. The offset is added non-circularly.
Write_Then_Circ_Add	Procedure. Writes the location pointed to by the current value of the iterator, then advances the iterator in accordance with the Step and BK specified in the initialization.
Write_Then_Circ_Sub	Procedure. Writes the location pointed to by the current value of the iterator, then advances the iterator in accordance with the Step and BK specified in the initialization.
Circ_Add	Procedure. Advances the iterator in accordance with the Step and BK specified in the initialization.
Circ_Sub	Procedure. Advances the iterator in accordance with the Step and BK specified in the initialization.
Circ_Iter_EB_Plus_Index	Extracts and returns the "EB+Index" part of the iterator.
Circ_Iter_Step	Extracts and returns the Step part of the iterator.
Circ_Iter_BK	Extracts and returns the BK part of the iterator.

Hints for Improved Object Code Quality:

- For improved code, initialize the "most important" iterators first in textual order in the source code.

- If all BK's of all active iterators are not provably the same at compile-time, generated code will degrade considerably.

- Always release iterators when they are no longer needed.

5.13.6. Bit-Reversed Addressing

The 320C30 bit-reversed addressing modes are made available through a set of generic functions that model the entire process with an iterator object and a set of subprograms to:

- Initialize the iterator.
- Read an object specified by the current value of the iterator.
- Advance the iterator to a new object.
- Release the iterator for later use.

These generics are documented using the same names and terms as in section 6.4 of the *TMS320C30 User's Guide*.

A fixed number of iterators are available for use in bit-reversed addressing. Iterators are named by the enumeration literals of the type Brev_Iterator_Name_Type. More than one iterator may be active at any given time.

The same rules regarding the use of circular addressing iterators apply to bit-reversed addressing iterators.

The functions that operate on bit-reversed iterators are:

Init_Brev_Iter	Procedure. Allocate and initialize an iterator. From left to right, the parameters are:
	Base_Addr_Plus_Start_Index Usually Array(Start_Index, 0)'address, assuming that the second dimension of the array holds the data points to be addressed between each change in the bit-reverse iterator. Array'address mod Two_To_N *must* = 0. This can be guaranteed only if Array is placed in memory using an Ada *address* clause.
	Two_To_N Usually Array'size/2. Must be a power of two.
	Name One of Brev_Iterator_Name_Type.
Release_Brev_Iter	Procedure. Releases the iterator resources to the compiler for other use.
Read_Brev_Iter	Returns the value pointed to by the current value of the iterator, plus some arbitrary integer offset.
Read_Then_Brev_Add	Returns the value pointed to by the current value of the iterator, then advances the iterator according to the Two_To_N specified in the initialization.
Write_Brev_Iter	Procedure. Writes the location pointed to by the current value of the iterator, plus some arbitrary integer offset.
Write_Then_Brev_Add	Procedure. Writes the location pointed to by the current value of the iterator, then advances the iterator in accordance with the Two_To_N specified in the initialization.

parameter(s) are actually read in many of the functions. However, in all cases the results are only accurate to single (32-bit) floating precision.

Figure 5-4 shows an example of the use of POWER-32 instantiated as the "**" operator for floats.

```
with intrinsics; use intrinsics;
with text_io;
with flt_io;

procedure test(a : float) is
    function "**" is new POWER_32(float,float);
    function ALOG10 is new ALOG10_32(float,float);
    b : float;
begin
    text_io.put_line("test: a, 10**a, alog10(10**a):");
    flt_io.put(a);
    b := 10.0**a;
    flt_io.put(b);
    flt_io.put(alog10(b));
    text_io.new_line;
end test;
```

Figure 5-4: Using the intrinsic Math Functions

A call to an instance of any of these results in a call to an extremely fast-executing function to perform the computation. These are "shared-code" generics in the sense that there will be only one object-code version of each function created no matter how many instantiations are made.

The code generator contains built-in knowledge that these function calls are free from side effects and thus do not cause optimizations to be blocked. The code generator also knows exactly which of the volatile registers are used by each routine and will not save active values from registers that are not used by the routine being called.

Algorithms for the routines were adapted from *Software Manual for the Elementary Functions*; Cody and Waite, Prentice Hall 1980; and *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*; Milton Abramowitz and Irene A. Stegun, National Bureau of Standards (Applied Mathematics Series 55), Washington D.C., 1964 (reprinted 1970); and the *TMS320C30 User's Guide*. Some algorithms were developed internally.

All routines are accurate to single (32-bit) floating precision. An augmented set of Cody-Waite accuracy tests have been used to test them. Loss of precision was found to be limited to 2 bits or less of the 24-bit mantissa for all the functions. Test results are available from Tartan on request.

Every attempt has been made to avoid raising an exception for any input value. Reasonable values are returned under all conditions. It is assumed that most signal processing applications work in a "press on" mode. In particular, the following rules hold for all routines, except the MEDIUM_FAST, FAST and QUAD1 variants of SIN and COS:

1. The legal range for each parameter is that defined by the intersection of:

    • The values of the parameter over which the pure mathematical function is defined

    • The set of inputs yielding functions results expressable within the C30's 32-bit floating point format

    • The set of inputs with the values representable by the C30's 40-bit floating point format

2. With the exception of the MEDIUM_FAST, FAST and QUAD1 variants of SIN and COS, any parameter not in legal range is replaced with the closest legal value on entry to the function. For the MEDIUM_FAST, FAST and QUAD1 variants of SIN and COS, the function result is undefined for parameters outside the restricted input range.

COSH	38 te<|x|<te69	79 |x|<= te	79 |x| <= te
TANH	53 |x|<=.549	82 .549<|x|<=9.01	75 |x| <= 2.196
COTH	86 |x|<=.549	115 .549|x|<=9.01	108 |x| <= 2.196
ASINH	19 |x|<=.5	93 |x| > .5	93 |x| <= 100
ACOSH	53 x >= 2**32	89 x < 2**32	89
ATANH	25 |x| <= .5	85 |x| > .5	55
ACOTH	54 |x| >= 2	85 |x| < 2	70

KEY:
====

```
tc   = 2**24 * PI - PI/2 = 52707176.96 ~= 52707000
ts   = 2**24 * PI        = 52707178.53 ~= 52707000
tt   = (2**24 -.5) * PI/2= 26353588.48 ~= 26353000
te   = 88.71875
te69 = 88.71875 + 0.69316 = 89.41191
```